

Mecklenburg County

Coding Standards for .NET ApplicationsRevision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description of Modifications** | **Updated By** |
| 08/19/04 | 0.1 | Initial Draft | Brian Sturgill |
| 09/21/04 | 0.2 | Added a bullet to “Use *Camel Case*” to the Variable Naming Conventions section. | Brian Sturgill |
| 09/21/04 | 0.3 | Deleted duplicate section 1.5. | Brian Sturgill |
| 09/21/04 | 0.4 | Added additional rules to the Variable Naming Conventions section to include module and local scoped variables. | Brian Sturgill |
| 09/23/04 | 0.5 | Added nDoc and in-line comment sections | Michael Piwonka |
| 09/27/04 | 0.6 | Added stored proc naming conventions | Michael Piwonka |
| 02/28/07 | 0.7 | Added code review standards. | Ken McWhirter |
| 12/16/11 | 1.1 | Added checklist references and security info. | Scott Newell |
| 09/22/15 | 2.0 | Reviewed / Updated | Elliot Faure Garcia |

Table of Contents

[Overview 5](#_Toc430695577)

[Standards 6](#_Toc430695578)

[Language Selection 6](#_Toc430695579)

[Code Generation 6](#_Toc430695580)

[Formatting 6](#_Toc430695581)

[In-line Comments 6](#_Toc430695582)

[Naming Standards 7](#_Toc430695583)

[1.1 Capitalization Styles 7](#_Toc430695584)

[1.2 Method Naming Guidelines 7](#_Toc430695585)

[1.3 Variable Naming Guidelines 7](#_Toc430695586)

[1.4 Class Naming Guidelines 8](#_Toc430695587)

[1.5 Parameter Naming Guidelines 8](#_Toc430695588)

[1.6 Constant Naming Guidelines 8](#_Toc430695589)

[Stored Procedure Naming Standards 8](#_Toc430695590)

[Practices 10](#_Toc430695591)

[General Programming Practices 10](#_Toc430695592)

[1.1 Variables 10](#_Toc430695593)

[1.2 Control Structures 10](#_Toc430695594)

[Exception Handling 10](#_Toc430695595)

[1.1 When to use exceptions 10](#_Toc430695596)

[1.2 Throwing and Catching Exceptions 11](#_Toc430695597)

[1.2.1 Catching and Re-Throwing Exceptions 12](#_Toc430695598)

[1.2.2 Custom Exceptions 12](#_Toc430695599)

[1.2.3 Creating a Custom Exception Class 13](#_Toc430695600)

[1.2.4 Adding additional properties to the exception class 13](#_Toc430695601)

[1.2.5 How to override base properties 14](#_Toc430695602)

[1.2.6 Serialization of Exception Data 14](#_Toc430695603)

[1.2.7 Some Exception Data 15](#_Toc430695604)

[Garbage Collection 16](#_Toc430695605)

[Code Review 17](#_Toc430695606)

[Option Strict and Explicit 17](#_Toc430695607)

[Frequent Code Paths 17](#_Toc430695608)

[Loops 17](#_Toc430695609)

[Resource Cleanup 17](#_Toc430695610)

[String Management 18](#_Toc430695611)

[Boxing 18](#_Toc430695612)

[Exceptions 18](#_Toc430695613)

[Collections 18](#_Toc430695614)

[Late Binding 18](#_Toc430695615)

[Properties 18](#_Toc430695616)

[Variable Scope 19](#_Toc430695617)

[Session State 19](#_Toc430695618)

[Optimize Web Pages by Reducing Page Size 19](#_Toc430695619)

[Page.IsPostBack 19](#_Toc430695620)

[ViewState 19](#_Toc430695621)

[Security 20](#_Toc430695622)

[References 21](#_Toc430695623)

[Books: 21](#_Toc430695624)

[Periodicals: 21](#_Toc430695625)

[General Web Sites: 21](#_Toc430695626)

[Performance: 21](#_Toc430695627)

[Exception Handling: 21](#_Toc430695628)

[Web Workshop Articles: 21](#_Toc430695629)

Overview

This document contains the coding standards and conventions to be used by teams developing .NET applications at Mecklenburg County. The goal of this document is to outline a common set of standards for creating source code that is easy to maintain without overburdening the developer with a large set of rigid rules. The conventions covered in this document include naming standards, commenting, and other best practices that have been identified sense .NET inception. Developers should apply these standards where they apply and to logically extend these standards to new situations that they encounter.

The general purpose of this guide is:

* Create consistent coding style to make it easier for developers to read and understand source code.
* Avoid common errors and unnecessary errors.
* Reduce code maintenance costs
* Create a standard basis for code reviews.
* Incorporate industry best practices
* Establish a common methodology

Many of the standards and practices outlined in this document are guidelines. Specific project needs may vary the use of these guidelines depending on the situation. This document is intended to be a living document that expands and contracts to incorporate new practices and the changing business needs and strategic goals of Mecklenburg County IST.

Standards

This section outlines the standards to be followed when developing .NET applications for Mecklenburg County. These standards are used to create a consist environment that promotes readability and maintainability of code.

Language Selection

C# is the preferred language for developing applications at Mecklenburg County.

Code Generation

Code generation tools include everything from simple wizards to Rational Rose generation of code from UML. Typically, code generation tools produce code that is poorly formatted, poorly documented, and poorly understood by the developer using the tools.

1. Avoid use of code generation tools that create code that will be manually extended or maintained.
2. If code generation tools are used, make sure generated code is formatted to the standards listed in this document.

Formatting

1. At some point in time, templates will be available for each language that will contain comment headers and the preferred file layout. These templates should be used when starting a new module or component.
2. If not template is available follow the styles outlined in this document. A new template should be created when work is started in a new language.
3. If maintaining existing code, if no discernable style is established, consider reformatting the style listed here. If a discernable style exists, prefer following the nuances of the existing code.
4. Don’t create lines longer than 80 characters in length. This line is difficult to print and makes it hard to understand using the horizontal scroll.
5. Use the default tab space setting.
6. Always indent the contents of methods, loops and logical blocks.
7. Use blank lines to separate and organize code.
8. Place a single space before and after each operator and control statement.
9. Place a single space after each common in a parameter list.

In-line Comments

Code that is used to fix bugs and code that is used to work around problems should always be commented. The comment style to be used is: [date] [username] [description] before the change and [date] [username] after the change.

// 01/01/2004 ABC Change to allow ...

Console.WriteLine("Here is some code ....")

// 01/01/2004 ABC

1. Comments should be indented at the same level as the code they are commenting.
2. Comments should not be placed at the end of a line (assembly style). End-of-line style is less visible and more difficult to vertically align. Exception to this rule is terse definition of variable which can be placed after the variable declaration.
3. Place a blank line before a comment to highlight the code.

Naming Standards

According to the *C# Coding Conventions (C# Programming Guide)* “Coding conventions serve the following purposes:

* They create a consistent look to the code, so that readers can focus on content, not layout.
* They enable readers to understand the code more quickly by making assumptions based on previous experience.
* They facilitate copying, changing, and maintaining the code.
* They demonstrate C# best practices.”

## Capitalization Styles

Use the following conventions for capitalizing identifiers:

**Pascal Case**

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. Pascal case can be used for identifiers of three or more characters. An example of Pascal case is: BackColor.

**Camel Case**

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. An example of camel case is: backColor.

**Uppercase**

All letters in the identifier are capitalized. Use this convention for identifiers that consist of two or fewer letters. An example is System.IO or System.Web.UI.

## Method Naming Guidelines

1. Use Pascal case.
2. Use verbs or verb phrases to correctly name methods.

The following are example of properly named methods:

Invoke()

ReverseText()

DataBind()

## Variable Naming Guidelines

1. Use *Camel Case*.
2. Use descriptive variable names.
3. Use variable names to describe the entity that the variable represents.
4. Use only alphanumeric characters in variable names.
5. Avoid use of Hungarian notation to denote underlying variable type.

According to Steve McConnell, “Hungarian notation ties variable type to variable name, which is contrary to the principles of data hiding and encapsulation. Using Hungarian notation distracts from the main purpose of the variable name, which is to describe what the data contained in the variable means. Hungarian notation also makes maintenance more difficult: when a variable type changes, each appearance of the variable name change.”

1. Prefix locally scoped variables with a “\_” (underscore) character.

## Class Naming Guidelines

1. Use a noun or noun phrase to name a class.
2. Use *Pascal Case*.
3. Use abbreviations sparingly.
4. Do not use type prefix, such as “c” for class, on a class name.
5. Do not use the underscore character.
6. Use compound words to name a derived class. The second part of the name should be the name of the base class. For example, ImmediateAlert is an appropriate name for a class derived from Alert.
7. Do not use class names as part of the naming of a class property.

Example:

**NO:** Report.ReportName

**YES:** Report.Name

## Parameter Naming Guidelines

1. Use *camel case* for parameter names.
2. Use descriptive parameter names.
3. Use names that describe the parameter not the type.
4. Do not use reserved words.

## Constant Naming Guidelines

1. Use upper-case alphanumeric characters in constant names; separate words within a constant name by underscores.
2. Use constants for multiple use text strings.
3. Use constants for single use text string as appropriate. Text strings that will be displayed to users should always be put into string constants or string tables. This allows them to be translated or updated as needed.

Stored Procedure Naming Standards

We will follow the naming convention we employed for the original ISSI project, except that we will include the word *ISSI* in the name to differentiate the new stored procedures written for the .NET version from the existing stored procedures. Consequently, the naming convention will look like this:

* usp\_select\_issi\_file
* usp\_delete\_issi\_file
* usp\_update\_issi\_file
* usp\_insert\_issi\_file

Following this standard will also keep the new stored procedures grouped by functionality within Query Analyzer/Enterprise Manager.

Practices

This section covers coding practices in general. These guidelines should help ensure that we are building applications that follow best programming practices used throughout the industry.

General Programming Practices

## Variables

1. Declare a variable as close as possible to its first use.
2. Initialize variables when declared.
3. Precede variable declarations with a brief comment describing the purpose more fully. Comments help to describe the purpose of a variable that even the most descriptive names cannot.

## Control Structures

1. Use descriptive names for loop indices. Name the loop index after the thing being iterated over, such as ‘row’ or ‘col’. If the loop is simple and small, traditional loop indices such as ‘i’, ‘j’, and ‘k’ are acceptable, although not preferred.
2. Prefer *for* loops where appropriate. *for* loops place all loop-control code in one place, which makes the loop more readable.

Exception Handling

## When to use exceptions

According to Jesse Liberty in “Programming C#”, “An exception is an object that encapsulates information about an unusual program occurrence.” Exceptions are not bugs nor are they errors, but rather unusual or unpreventable circumstances that you want to ensure that your program gracefully handles.

Throwing exceptions in .NET is expensive and should not be used for logic flow (or return codes) in your program. In addition to that if you have an event that happens on a regular basis, you would want to use logic to check for that event rather than have it fall to the catch statement.

**C#:**

if (Conn.State != ConnectionState.Closed)

Conn.Close();

As opposed to:

try

{

Conn.Close();

}

catch(Exception e)

{

//Do something here.

}

finally

{

//Clean up here

}

## Throwing and Catching Exceptions

To indicate an abnormal[[1]](#footnote-1) condition in code you will use the “throw” keyword.

**C#:**

[From MSDN help](ms-help://MS.VSCC/MS.MSDNVS/csref/html/vclrfTheThrowStatement.htm):

**Example**

This example demonstrates how to throw an exception using the throw statement.

// throw example

using System;

public class ThrowTest

{

public static void Main()

{

string s = null;

if (s == null)

{

throw(new ArgumentNullException());

}

Console.Write("The string s is null"); // not executed

}

}

**Output**

The following exception occurs:

System.ArgumentNullException

The catch clause is used to specify handlers for exceptions that have been thrown. There can be multiple catch statements for each type of exception you wish to catch. It is recommended to start with the more specific exceptions and then work your way to more general exceptions. This ensures the more specific exception is caught prior to the general exception.

**C#:**

[From MSDN Help:](ms-help://MS.VSCC/MS.MSDNVS/csref/html/vclrfTheTrycatchStatement.htm)

**Example**

In this example, two catch statements are used. The most specific exception, which comes first, is caught.

// Ordering catch clauses

using System;

class MyClass

{

public static void Main()

{

MyClass x = new MyClass();

try

{

string s = null;

x.MyFn(s);

}

// Most specific:

catch (ArgumentNullException e)

{

Console.WriteLine("{0} First exception caught.", e);

}

// Least specific:

catch (Exception e)

{

Console.WriteLine("{0} Second exception caught.", e);

}

finally

{

//Clean up all resources here. E.g. DB connections, etc.

}

}

public void MyFn(string s)

{

if (s == null)

throw new ArgumentNullException();

}

}

**Output**

The following exception occurs:

*System.ArgumentNullException*

In the preceding example, if you start with the least specific catch clause, you will get the error message:

*A previous catch clause already catches all exceptions of this or a super type ('System.Exception')*

However, to catch the least specific exception, replace the throw statement by the following one:

*throw new Exception();*

### Catching and Re-Throwing Exceptions

If you catch an exception and you wish to re throw it, you need to add the original exception as the inner exception, in order to keep stack trace information.

**C#:**

catch (exception e)

{  
     throw new exception(e);  
}

### Custom Exceptions

You should create a custom exception when you need to:

1. Add custom or unique information to the exception.

E.g. Properties – UserID for a Database Connection.

1. If you need to add special functionality to the exception object.

E.g. Formatting the message value to meet a specific format or localization of a string.

### Creating a Custom Exception Class

Your base class will have at least three constructors. It is required to end your custom exception class in “Exception”. E.g. DBException

**C#:**

public DBException()

public DBException (string message)

public DBException (string message, Exception inner)

The first two constructors are used to create exceptions to be thrown, with the second one used the majority of the time.

The third constructor is used to wrap exceptions with more information. In most exception classes, these constructors merely forward their parameters to the base class constructor. For example, code the third constructor as follows:

**C#:**

public DBException (string message, Exception inner): base(message, inner)

{

}

### Adding additional properties to the exception class

To add additional fields to the base exception class you must do five things.

1. Add the field to the class
2. Add a read-only property to access the fields value

**C#:**

public class DBException: ApplicationException, ISerializable

{

private string name;

…

public string Name;

{

get

{

return name;

}

}

}

1. Add a constructor to take the new value

**C#:**

public DBException ( string message, string name) : base(message)

{

this.name = name;

}

1. Override the GetObjectData() method to serialize the additional field(s)

**C#:**

public override void GetObjectData(SerializationInfo info, StreamingContext context)

{

base.GetObjectData(info, context);

info.AddValue(“Name”, name);

}

Now within the catch block for your newly defined Exception class the Name will be an available property.

1. Override the message property. Please see the following section.

### How to override base properties

To override base Exception properties is very straight forward and is useful in situations where you want to have a custom look to things such as error messages.

**C#:**

public override Message

{

get

{

//This will prepend the Exception Name and it’s value to the

//Message.

string myName = String.Format(“Exception Name: {0}”, name);

return myName + “ ” + base.Message;

}

}

### Serialization of Exception Data

You will also want to enable Serialization of your custom exception class so it can easily travel across machine boundaries. This is usually possible by adding the [Serializable] attribute to your class. But you will need to add the deserialization constructor to extract custom values because you have implemented the ISerializable interface.

**C#:**

Setting up the class for the new field to be serialized.

public override void GetObjectData(SerializationInfo info,

StreamingContext context)

{

base.GetObjectData(info, context);

//Add all additional custom fields

info.AddValue("Name", name);

}

Strictly speaking, this marks our class as re-implementing the interface as it already inherits an implementation from the base class. This function first calls the base function to make sure it gets a chance to save its data, and then saves the value. The string "Name" is passed in as a key to be associated with the field we saved, and we can use the same key to extract the value later.

**Please Note:** The name GetObjectData() is a bit confusing. In short, it's called by the serializer to get the object data, thus the name.

Next, we need to modify our deserialization constructor so that it extracts that value. That's simply a matter of extracting the value for our key.

**C#:**

public DBException(SerializationInfo info, StreamingContext context):

base(info, context)

{

name = info.GetInt32("Name");

//Additional custom fields

}

### Some Exception Data

All classes are located in the System namespace unless otherwise noted.

|  |  |
| --- | --- |
| **Data** | **Source** |
| Date and time of exception | DateTime.Now |
| Machine name | Environment.MachineName |
| Exception source | Exception.Source |
| Exception type | Type.FullName obtained from Object.GetType |
| Exception message | Exception.Message |
| Exception stack trace | Exception.StackTrace—this trace starts at the point the exception is thrown and is populated as it propagates up the call stack. |
| Call stack | Environment.StackTrace—the complete call stack. |
| Application domain name | AppDomain.FriendlyName |
| Assembly name | AssemblyName.FullName, in the System.Reflection namespace |
| Assembly version | Included in the AssemblyName.FullName |
| Thread ID | AppDomain.GetCurrentThreadId |
| Thread user | Thread.CurrentPrincipal in the System.Threading namespace |

Garbage Collection

Garbage collection in the .NET runtime largely frees developers from having to worry about memory management. However, there are times when the default behavior of the .NET garbage collector is not optimal. Also, it is very important to understand how .NET garbage collection works in order to make proper use of the Close, Dispose, or Finalize methods that enable the developer to ensure that resources are properly torn down.

Most of the concern over .NET garbage collection has centered on what is being referred to as *non-deterministic finalization.* This is a very opaque way to say that developers can write cleanup code for tearing down resources, but have no control over when the actual teardown occurs. Instead, the garbage collector makes periodic tours of the managed heap, looking for objects with no application references. For each of these, it checks to see if a Finalize method exists. If it does, the garbage collector calls it prior to freeing the object’s memory. There are several issues related to this. A good discussion on garbage collection can be found in an MSDN article called

[Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework](http://msdn.microsoft.com/msdnmag/issues/1100/GCI/GCI.asp) [[2]](#footnote-2)

by Jeffrey Richter. A critical point is that the Finalize method might be mistaken for the equivalent of a C++ destructor or a VB terminate event, but is quite different. In fact, it is best to avoid use of the Finalize method. For objects that require cleanup prior to teardown, you will need to implement Finalize, but you should also implement a Close or Dispose method that sets the SuppressFinalize flag so that you can determine when the cleanup is done, at the same time instructing the garbage collector to skip the call to the object’s Finalize method. There are quite a few complexities involved – please see the article for details.

Code Review

Code reviews should be a regular part of your development process. Performance and scalability code reviews focus on identifying coding techniques and design choices that could lead to performance and scalability issues. The review goal is to identify potential performance and scalability issues before the code is deployed. The cost and effort of fixing performance and scalability flaws at development time is far less than fixing them later in the product deployment cycle.

## Option Strict and Explicit

Review your code and ensure that the **Strict** and **Explicit** options are turned on. This ensures that all narrowing type coercions must be explicitly specified. This protects you from inadvertent late binding and enforces a higher level of coding discipline. **Option** **Explicit** forces you to declare a variable before using it by moving the type-inference from run time to compile time.

## Frequent Code Paths

Prioritize your code review process by identifying code paths that are frequently executed and begin your review process in these areas.

## Loops

Even the slightest inefficiency inside a loop is magnified many times over depending on the number of iterations. Specifically watch out for repetitive property access inside your loops, using **foreach** instead of **for**, performing expensive operations within your loops, and using recursion. Recursion incurs the overhead of having to repeatedly build new stack frames. Using a **for** loop instead of **foreach** increases performance for iterating through .NET Framework collections that can be indexed with an integer.

**When looping:**

* Move any code out of the loop that does not change the loop.
* Avoid calling properties within loops and if you can, check what the property accessor does. Calling a property can be a very expensive operation if the property is performing complex operations.
* Avoid calling properties within loops and if you can, check what the property accessor does. Calling a property can be a very expensive operation if the property is performing complex operations.

Check if your code uses exceptions inside loops. This should be avoided. If you need to catch an exception, place the **try**/**catch** block outside the loop for better performance.

## Resource Cleanup

Make sure your code calls **Dispose** (or **Close**) on disposable resources. Make sure your code uses **finally** blocks or **using** statements to ensure resources are closed even in the event of an exception.

Check that your code calls **Dispose** or **Close** on all classes that support these methods. Common disposable resources include the following:

Database-related classes: **Connection**, **DataReader** and **Transaction**.

File-based classes: **FileStream** and **BinaryWriter**.

Stream-based classes: **StreamReader**, **TextReader**, **TextWriter**, **BinaryReader** and **TextWriter**.

## String Management

Excessive string concatenation results in many unnecessary allocations, creating extra work for the garbage collector. Use **StringBuilder** for complex string manipulations and when you need to concatenate strings multiple times.

## Boxing

Boxing causes a heap allocation and a memory copy operation. Review your code to identify areas where implicit boxing occurs. Pay particular attention to code inside loops where the boxing overhead quickly adds up. Avoid passing value types in method parameters that expect a reference type. Sometimes this is unavoidable. In this case, to reduce the boxing overhead, box your variable once and keep an object reference to the boxed copy as long as needed, and then unbox it when you need a value type again.

## Exceptions

You should catch exceptions for very specific reasons, because catching generally involves rethrowing an exception to the code that calls you. Rethrowing an exception is as expensive as throwing a new exception. Check that when your code catches an exception, it does so for a reason.

When re-throwing exceptions, be sure to wrap the original exception as an inner exception; this makes debugging of code easier.

Use the overloaded **Server.Transfer** method **Server.Transfer(String,bool)** instead of **Server.Transfer**, **Response.Redirect**, and **Response.End** to avoid exceptions.

## Collections

Storing value types in a collection involves a boxing and unboxing overhead. The overhead can be significant when iterating through a large collection for inserting or retrieving the value types. Consider using arrays or developing a custom, strongly typed collection for this purpose.

If you use a **Hashtable**:

* **Do you store small amounts of data in a Hashtable**?

If you store small amounts of data (10 or fewer items), this is likely to be slower than using a **ListDictionary**. If you do not know the number of items to be stored, use a **HybridDictionary**.

* **Do you store strings**?

Prefer **StringDictionary** instead of **Hashtable** for storing strings, because this preserves the string type and avoids the cost of up-casting and down-casting during storing and retrieval.

## Late Binding

In C#, a variable is late bound if it is declared as an **Object** or is without an explicit data type. When your code accesses members on late-bound variables, type checking and member lookup occurs at run time. As a result, early-bound objects have better performance than late-bound objects.

## Properties

You can expose class-level member variables by using public fields or public properties. The use of properties represents good object-oriented programming practice because it allows you to encapsulate validation and security checks and to ensure that they are executed when the property is accessed.

Properties should be simple and should not contain more code than required for getting/setting and validation of the parameters. Properties can look like inexpensive fields to clients of your class, but they may end up performing expensive operations.

## Variable Scope

You can scope member variables as either public or private members. Think carefully about which members should be made public because with public members you run the risk of exposing sensitive data that can easily be manipulated.

## Session State

Avoid storing complex objects in session state, particularly if you use an out-of-process session state store. When using out-of-process session state, objects have to be serialized and deserialized for each request, which decreases performance.

## Optimize Web Pages by Reducing Page Size

Try to keep the page size to a minimum. Large page sizes place increased load on the CPU because of increased processing and a significant increase in network bandwidth utilization, which may lead to network congestion. Both of these factors lead to increased response times for clients. Consider the following guidelines to help reduce page size:

Use script includes (script tags rather than interspersing code with HTML).

Remove redundant white space characters from your HTML.

Disable view state for server controls where it is not needed.

Avoid long control names.

Minimize the use of graphics, and use compressed images.

Consider using cascading style sheets to avoid sending the same formatting directives to the client repeatedly.

## Page.IsPostBack

Check that the logic in your page uses the **Page.IsPostBack** property to reduce redundant processing and avoid unnecessary initialization costs. Use the **Page.IsPostBack** property to conditionally execute code, depending on whether the page is generated in response to a server control event or whether it is loaded for the first time.

## ViewState

Evaluate each page to determine if you need view state enabled. View state adds overhead to each request. The overhead includes increased page sizes sent to the client as well as a serialization and deserialization cost. You do not need view state under the following conditions:

The page does not post back to itself; the page is only used for output and does not rely on response processing.

Your page's server controls do not handle events and you have no dynamic or data-bound property values (or they are set in code on every request).

If you are ignoring old data and repopulating the server control every time the page is refreshed.

## Security

There are many security considerations to take into account when coding .NET applications. Applications and Technology team should follow guidelines outlined in [Microsoft’s Secure Coding Guidelines](https://msdn.microsoft.com/en-us/library/8a3x2b7f(v=vs.110).aspx).

References

Books:

* Barwell and Blair. *Professional VB.NET*. Birmingham, UK: Wrox Press, 2001.
* Kernighan and Plauger. *The Elements of Programming Style*. New York: McGraw-Hill, 1982.

Periodicals:

Visual Studio Magazine

MSDN Magazine

XML Magazine

General Web Sites:

[The Basics of Programming Model Design](http://msdn.microsoft.com/library/techart/msdn_basicpmd.htm)

<http://www.gotdotnet.com/>

<http://www.codehound.com/>

<http://www.codeproject.com/>

<http://www.gotdotnet.com/quickstart/howto/>

<http://msdn.microsoft.com/msdnmag/issues/01/07/vbnet/vbnet.asp>

<http://www.construx.com/cxone>

Performance:

[Performance Tips and Tricks in .NET Applications](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftips.asp#_blank) by Emmanuel Schanzer (Microsoft Corporation August 2001)

[Performance Considerations for Run-Time Technologies in the .NET Framework](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftechs.asp) by Emmanuel Schanzer (Microsoft Corporation August 2001)

[Improving .NET Application Performance and Scalability](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftechs.asp) by J.D. Meier, Srinath Vasireddy, Ashish Babbar, Rico Mariani, and Alex Mackman (Microsoft Corporation May 2004)

Exception Handling:

[Exception Management in .NET](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp), by Kenny Jones and Edward Jezierski (Microsoft Corporation August 2001)

[The Well-Tempered Exception](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp08162001.asp), by Eric Gunnerson (Microsoft Corporation August 2001)

Web Workshop Articles:

MSDN Online Design Area: <http://msdn.microsoft.com/workshop/design/default.asp>

ASP Conventions: <http://msdn.microsoft.com/workshop/server/asp/aspconv.asp>

15 ASP Tips to Improve Performance and Style: <http://msdn.microsoft.com/workshop/server/asp/ASPtips.asp>

ASP Guidelines: <http://msdn.microsoft.com/workshop/server/asp/server122799.asp>

HTML Tips, Tricks, and Secrets: <http://msdn.microsoft.com/workshop/author/html/mytips.asp>

1. Also known as Abbey Normal. [↑](#footnote-ref-1)
2. http://msdn.microsoft.com/msdnmag/issues/1100/GCI/GCI.asp [↑](#footnote-ref-2)